Spring MVC:

===============

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

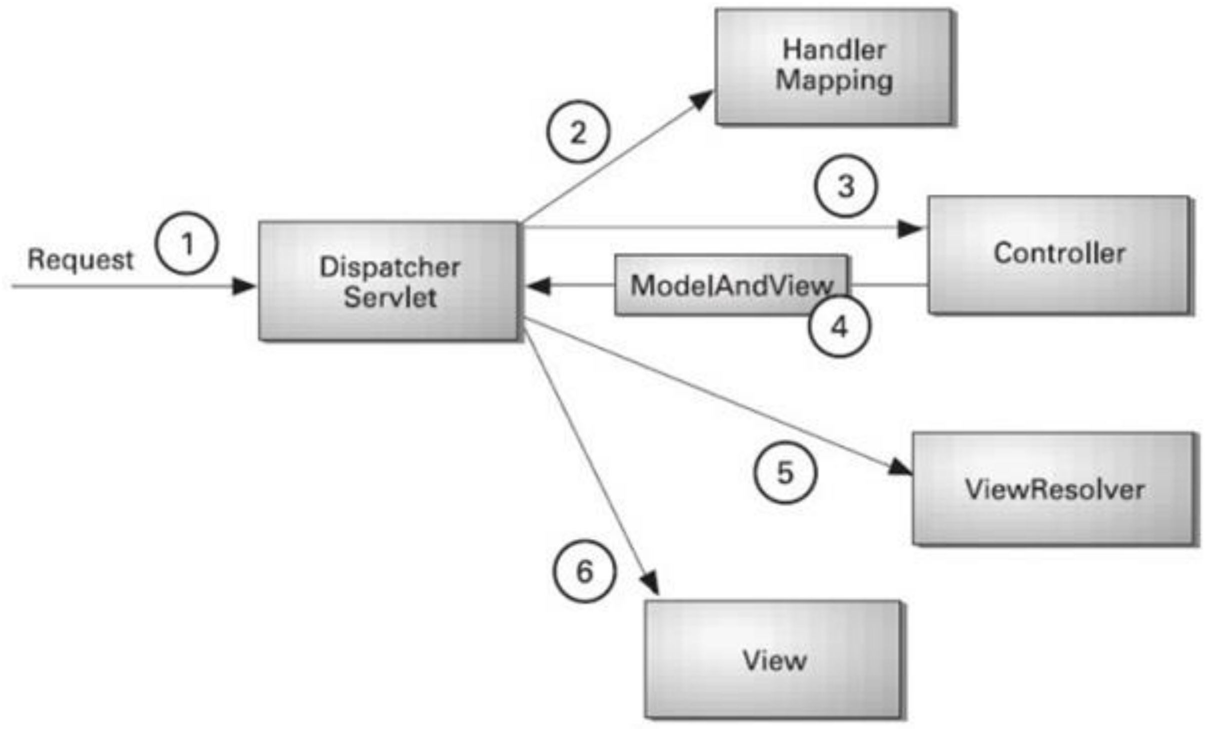
Model - A model contains the data of the application. A data can be a single object or a collection of objects.

Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

MVC Flow



1.After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

2.The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

3.The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

4.Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

Advantages of Spring MVC

=========================

1. Light Weight: It uses light-weight servlet container to develop and deploy your application.
2. Speed development: As we are suing the layred arichetecture its easy to develop the code in faster manner
3. Separate roles - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
4. Reusable business code - Instead of creating new objects, it allows us to use the existing business objects.
5. Easy to test - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.

Model Interface:

****

In Spring MVC, the model works a container that contains the data of the application. Here, a data can be in any form such as objects, strings, information from the database, etc.

It is required to place the Model interface in the controller part of the application. The object of HttpServletRequest reads the information provided by the user and pass it to the Model interface. Now, a view page easily accesses the data from the model part.

Methods:

============

addAllAttributes(Collection<?> arg): It adds all the attributes in the provided Collection into this Map.

mergeAttributes(Map<String,?> arg): It adds all attributes in the provided Map into this Map, with existing objects of the same name taking precedence.

boolean containsAttribute(String arg): It indicates whether this model contains an attribute of the given name

How to render the mode data in JSP page:

****

We can add the jstl core tages dependent jar liberary in jar.

Ex:

```
<body>

<h1>Start here</h1>

${greeting}

</body>
```

How to render the mode data in HTML page:

****

The Spring MVC form tags can be seen as data binding-aware tags that can automatically set data to Java object/bean and also retrieve from it. Here, each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and easy to use.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Input field:

****

```
<form:input type=?email? path="email" />
```

Displaying the Data in HTML page:

****

```
<body>

<p>Your reservation is confirmed successfully. Please, re-check the details.</p>

First Name : ${reservation.firstName} <br>

Last Name : ${reservation.lastName}

</body>
```

==
===============================

Spring All Annotations:

======================

Spring Core Framework Annotations:

****

@Required:

This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The @Required annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type BeanInitializationException is thrown.

@Autowired:

This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.

@Qualifier:

This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.This annotation is used along with @Autowired annotation.

If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException

Consider an example where an interface BeanInterface is implemented by two beans BeanB1 and BeanB2.

```
@Component
public class BeanB1 implements BeanInterface {
  //
}
@Component
public class BeanB2 implements BeanInterface {
  //
}
```

Now if BeanA autowires this interface, Spring will not know which one of the two implementations to inject.

One solution to this problem is the use of the @Qualifier annotation.

```
@Component
public class BeanA {
  @Autowired
  @Qualifier("beanB2")
  private BeanInterface dependency;
  ...
}
```

With the @Qualifier annotation added, Spring will now know which bean to autowire where beanB2 is the name of BeanB2.

@Configuration:

================

It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

```
@Configuration

public class Vehicle

{

@BeanVehicle engine()

{

return new Vehicle();

}

}
```

@ComponentScan:

=====================

It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

```
@ComponentScan(basePackages = "com.javatpoint")

@Configuration

public class ScanComponent

{

// ...

}
```

@Bean:

======================

It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

Example

```
@Bean

public BeanExample beanExample()

{

return new BeanExample ();

}
```

@Lazy:

==========

This annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use @Lazy annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on @Configuration classes. This indicates that all @Bean methods within that @Configuration should be lazily initialized.

@Value:

===========

This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation o inject values from a property file into a bean's attribute.

//
//

Spring Framework Stereotype Annotations

****

@Component:

=============

It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

@Service:

===============

It is also used at class level. It tells the Spring that class contains the business logic.It will perform calculations and call external APIs

@Repository:

================

It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

//
//

Spring MVC and REST Annotations

===

@Controller

===============

The @Controller annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC.

@RequestMapping:

===================

This annotation is used both at class and method level. It is used to map the web requests.

@GetMapping

===============

This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

@PostMapping

===============

This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

@PutMapping

===============

This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT)

@PatchMapping

===============

This annotation is used for mapping HTTP PATCH requests onto specific handler methods. @PatchMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH)

@DeleteMapping

===============

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

@ExceptionHandler

==================

This annotation is used at method levels to handle exception at the controller level. The @ExceptionHandler annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception.

@Mappings and @Mapping

======================

This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation. This annotation accepts an array of @Mapping having the source and the target fields.

@PathVariable:

================

It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.

@RequestParam:

====================

It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

@RequestHeader:

================

It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

@RequestAttribute:

====================

It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

@RequestPart

================

This annotation is used to annotate request handler method arguments. The @RequestPart annotation can be used instead of @RequestParam to get the content of a specific multipart and bind to the method argument annotated with @RequestPart. This annotation takes into consideration the "Content-Type" header in the multipart(request part).

@ResponseBody:

================

It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

@RequestBody:

==================

This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to object.

@ResponseStatus

===================

This annotation is used on methods and exception classes. @ResponseStatus marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with @ResponseStatus which is then inherited by all @RequestMapping methods.

@ControllerAdvice

===================

This annotation is applied at the class level. As explained earlier, for each controller you can use @ExceptionHandler on a method that will be called when a given exception occurs. But this handles only those exception that occur within the controller in which it is defined. To overcome this problem you can now use the @ControllerAdvice annotation. This annotation is used to define @ExceptionHandler, @InitBinder and @ModelAttribute methods that apply to all @RequestMapping methods. Thus if you define the @ExceptionHandler annotation on a method in @ControllerAdvice class, it will be applied to all the controllers.

@RestController

=====================

This annotation is used at the class level. The @RestController annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add @ResponseBody to all the RequestMapping method. It means that you no more use view-resolvers or send html in response. You just send the domain object as HTTP response in the format that is understood by the consumers like JSON.

@RestController is a convenience annotation which combines @Controller and @ResponseBody.

@RestControllerAdvice

=========================

This annotation is applied on Java classes. @RestControllerAdvice is a convenience annotation which combines @ControllerAdvice and @ResponseBody. This annotation is used along with the @ExceptionHandler annotation to handle exceptions that occur within the controller.

\\
\\\

Spring Framework DataAccess Annotations

****

@Transactional:

====================

When you annotate @Transactional Spring loads your bean definitions, and has been configured to look for @Transactional annotations, it will create these proxy objects around your actual bean.

If you are using JPA call then all commits are with in this transaction boundary.

Lets say you are saving entity1, entity2 and entity3. Now while saving entity3 an exception occur, then as enitiy1 and entity2 comes in same transaction so entity1 and entity2 will be rollback with entity3.

\\
\\

Cache-Based Annotations

==============================

@Cacheable:

=============

This annotation is used on methods. The simplest way of enabling the cache behaviour for a method is to annotate it with @Cacheable and parameterize it with the name of the cache where the results would be stored.

@Cacheable("addresses")

public String getAddress(Book book){...}

In the snippet above , the method getAddress is associated with the cache named addresses. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

@CachePut

===============

This annotation is used on methods. Whenever you need to update the cache without interfering the method execution, you can use the @CachePut annotation. That is, the method will always be executed and the result cached.

@CachePut("addresses")

public String getAddress(Book book){...}

Using @CachePut and @Cacheable on the same method is strongly discouraged as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict

===============

This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes you may want remove some cache data so that you can populate the cache with some fresh values. In such a case use the @CacheEvict annotation.

```
@CacheEvict(value="addresses", allEntries="true")

public String getAddress(Book book){...}
```

Here an additional element allEntries is used along with the cache name to be emptied. It is set to true so that it clears all values and prepares to hold new data.

\\\

Task Execution and Scheduling Annotations

****

@Scheduled

=============

This annotation is a method level annotation. The @Scheduled annotation is used on methods along with the trigger metadata. A method with @Scheduled should have void return type and should not accept any parameters.

There are different ways of using the @Scheduled annotation:

```
@Scheduled(fixedDelay=5000)

public void doSomething() {

  // something that should execute periodically

}
```

In this case, the duration between the end of last execution and the start of next execution is fixed. The tasks always wait until the previous one is finished.

In this case, the beginning of the task execution does not wait for the completion of the previous execution.

```
@Scheduled(initialDelay=1000,fixedRate=5000)

public void doSomething() {

 // something that should execute periodically after an initial delay

}
```

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async:

===========

@Async

This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Async can take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and the methods that return a value. However methods with return value must have a Future typed return values.

\\\
\\

Spring Boot Annotations:

****

@SpringBootApplication:

============================

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in com.example then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

The @SpringBootApplication is a convenient annotation that adds all the following:

@Configuration

@EnableAutoConfiguration

@ComponentScan